

TOSHIBA

Leading Innovation >>>

e-learning

C言語入門

C 言語入門

- コース概要

このコースでは、マイコンで使用する組み込み用のプログラムを作成する準備として、C言語の基礎知識を学習します。

- 講座内容

- C言語とは
- プリプロセス
- 変数
- 関数
- 演算子
- 制御文

C言語とは

C言語とは

プログラミング言語

C言語はプログラミング言語の一種です。

マイコンのCPUは2進数で表記される機械語しか理解できませんが、人間がプログラムを記述しやすくするために、C言語やアセンブリ言語などの「プログラミング言語」が用意されています。

C言語	ハードウェアを動かす命令を直接記述しないので、2000年以前は高水準言語と呼ばれていましたが、最近ではポインタでメモリアドレスを操作したり、I/Oなどを直接操作できる場合もあるため、中水準言語とも呼ばれています。
アセンブリ言語	ハードウェアを動かす命令を直接記述するので、低水準言語と呼ばれています。

C言語を使用してプログラムを記述した場合、プログラムをマイコンが理解することの出来る言語である「機械語」に変換する必要があり、これを「コンパイル」といいます。

C言語とは

C言語とアセンブリ言語の比較

C言語

C言語は
移植性が高い

文法はCPUに関係なく共通
ISOとANSIが協同で
Cの規格の標準化

C言語は
保守しやすい

構造化
手順を入れ子構造で
記述できるので、見やすい

セミicolon(;)までが1文
複数行に渡って記述できる

あいまい
記述した通りに動作するとは
限らない

アセンブリ言語

文法はCPU毎に固有
CPUメーカーなどが、
個々に決定

非構造化
分岐命令の多用により、
スパゲッティ構造になりやすい

1行に1命令
複数行に渡っては
記述できない

厳密
記述した通りに動作する

C言語とは

C言語のイメージ

```
/* これはC言語のイメージです */  
#include <stdlib.h>  
. . .  
. . .  
int data;  
void func(void)  
{  
    int i;  
    . . .  
    for ( i = 0 ; i < 4 ; i++ )  
    {  
        . . .  
        . . .  
        data = data+3;  
    }  
    . . .  
}
```

コメント

プログラムに注釈を記述するために使用

プリプロセッサ指令

コンパイルを行う前に、他のファイルの取り込みやテキストの置き換えなどを行う '#' から始まる命令

変数

メモリ上で扱うデータに名前をつけたもの
配列、ポインタ、構造体などがある

関数

CPUの動作手順を記述したもの
ある程度の大きさでまとめておき、
呼び出すことで再利用する

制御文

分岐、ループなど、実行の流れを記述する文

式

変数と演算子を組み合わせて、実行させる命令の内容を記述する

C言語とは

コンパイル

コンパイルとは、C言語で書かれたファイルからオブジェクトへ変換する処理です。

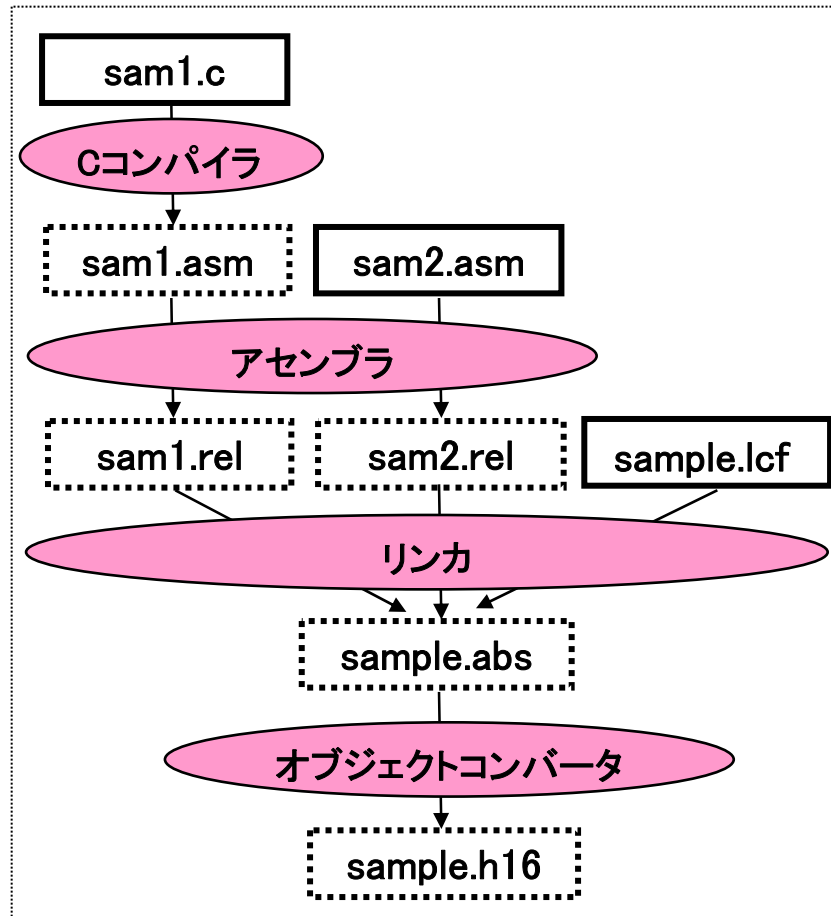
C言語ソースファイル
(***.c)

アセンブリ言語ファイル
(***.asm)

リロケータブル
オブジェクトファイル
(***.rel)

アブソリュート
オブジェクトファイル
(***.abs)

機械語
オブジェクトファイル
(***.h16)



※左記は弊社ツールの例です。

Cソースファイルから一旦アセンブリ言語に変換した後、アドレスを決定するリンク処理を経て、機械語のオブジェクトファイルへ変換します。

プリプロセス

プリプロセス

Cコンパイラが、ソースファイルをコンパイルする前に、行う処理のことをプリプロセスといいます。

プリプロセスには、以下のような機能(プリプロセッサ指令)があります。

(1)ファイルの読み込み (#include)

(2)テキスト文字列の置き換え (#define)

(3)条件によるソースコードの部分的選択 (#if)

プリプロセス

ファイルの読み込み

#include プリプロセッサ指令が記述された位置に、ファイル名で指定されたヘッダファイルを読み込みます。

通常ヘッダファイルには、複数のソースファイルで共通に使用する宣言を記述しておきます。

これにより、全てのソースファイルを一つ一つ手直ししなくても、ヘッダファイルを修正するだけで、必要な修正をきれなく行うことができます。

```
#include "ファイル名"
```

```
void func(void)  
{  
  
  
}
```

ここにファイルを読み込む

プリプロセス

テキスト文字列の置き換え

#define プリプロセッサ指令に従って、ソースファイルから特定のテキストを検索し、置換します。

これにより、同じ意味を持つ数値を意味のある文字列で表現したり、同じ意味を持つ数値を一度に修正することができます。

```
#define VAL_MIN 800
```

```
int data;
```

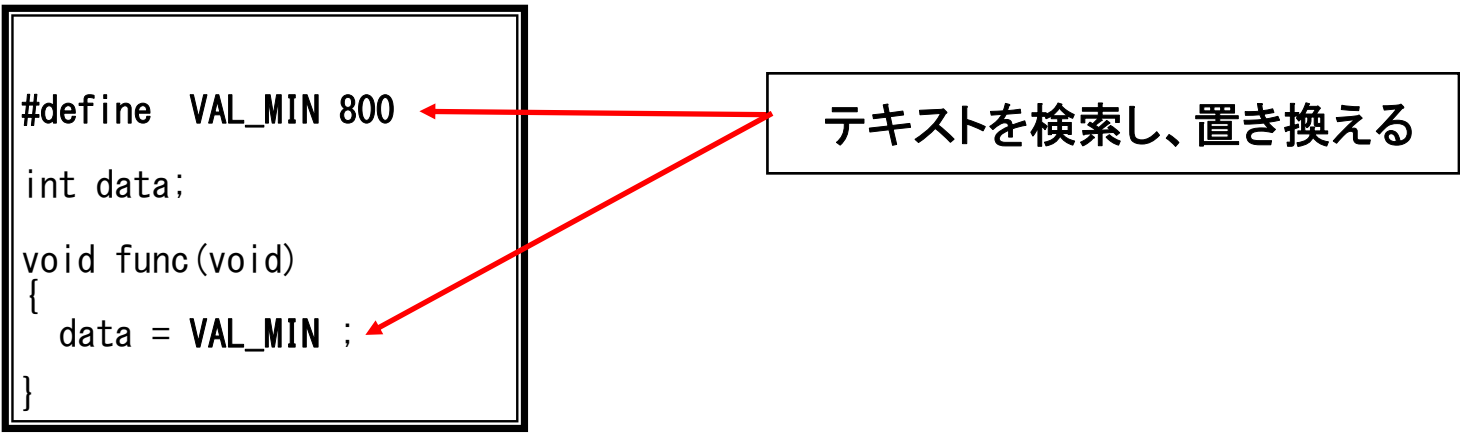
```
void func(void)
```

```
{
```

```
    data = VAL_MIN ;
```

```
}
```

テキストを検索し、置き換える

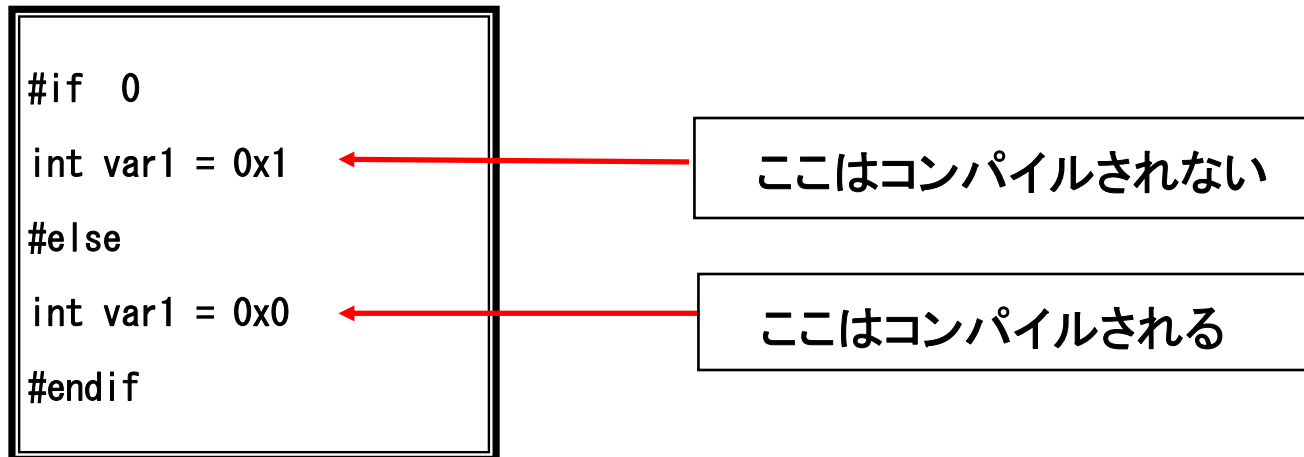
A diagram illustrating the text replacement process. On the left, a code block contains a preprocessor directive and a function definition. Two red arrows originate from a box on the right labeled 'テキストを検索し、置き換える' (Search and replace text). One arrow points to the '#define VAL_MIN 800' line, and the other points to the 'data = VAL_MIN ;' line within the function.

プリプロセス

条件によるソースコードの部分選択

#if プリプロセッサ指令に従って、コンパイルを行う部分と行わない部分を、コンパイルする際の条件によって、選択します。

一時的にコンパイルを抑止する場合や、同じソースファイルから複数の種類のオブジェクトを生成したい場合に使用します。



変数

変数

変数とは、プログラムで扱うデータに固有の名前を付けたものです。

一般的には変数の名前を「識別子」といい、変数が現しているデータを「値」といいます。

プログラムで変数を使用できるようにするためには、あらかじめ名前とどのようなデータであるかを明示する必要があり、これを「**変数の宣言**」といいます。

変数の宣言は以下のキーワードの組み合わせで行います。

- 1) 「記憶クラス指定子」
- 2) 「型修飾子」
- 3) 「型指定子」
- 4) 「識別子」

変数宣言の形式

記憶クラス指定子	型修飾子	型指定子	識別子;
----------	------	------	------

変数

記憶クラス指定子(1)

記憶クラス指定子は、その変数の「スコープ(scope)」と「記憶域期間(storage duration)」を指定するキーワードです。

スコープとは、その変数が見える(使用できる)範囲であり、

記憶域期間とは、その変数が消えずに残っている範囲です。

スコープ

```
void func1(void)
{
    int    data;
}

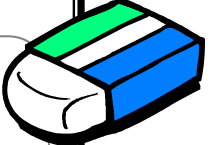
void func2(void)
{
    /*ここからはdata見えない*/
}
```



記憶域期間

```
void func1(void)
{
    int    data;
}

void func2(void)
{
    /*ここではdataは消えている*/
}
```



変数

記憶クラス指定子(2)

「スコープ」とは、その変数ができる範囲です。イメージは以下のようになります。

<scope1.c>

```
int    global_data;  
static int  static_data1;  
  
void func1(void)  
{  
  
    static int  static_data2;  
    int        auto_data;  
  
}  
  
void func2(void)  
{  
  
}
```

グローバルスコープ
記憶クラス指定子無し
関数などのブロックの外で宣言
→すべての領域で使用可能

ファイルスコープ
記憶クラス指定子はstatic
関数などのブロック外で宣言
→ファイル内でのみ使用可能

ローカルスコープ
記憶クラス指定子は何でも良い
関数などのブロック内で宣言
→ブロック内でのみ使用可能

<scope2.c>

```
void func3(void)  
{  
  
}
```



変数

記憶クラス指定子(3)

「記憶域期間」とは、その変数が残っている範囲です。イメージは以下のようになります。

<life1.c>

```
int    global_data;  
static int  static_data1;  
  
void func1(void)  
{  
    static int  static_data2;  
    int    auto_data;  
}  
  
void func2(void)  
{  
}
```

静的変数

関数などのブロックの外で宣言
または、static付きで宣言
→すべての領域で変数が残る

動的変数

関数などのブロックの中で宣言
かつ、staticが付いていない
→ブロック終了とともに消える

<life2.c>

```
void func3(void)  
{  
}
```



変数

記憶クラス指定子(4)

auto	変数宣言を行ったブロック内のみで有効になる変数です。関数が終了するなど、ブロックが終わると消えます。
static	autoとは異なり関数が終了しても消えません。再び同じ関数を呼び出した場合は前回の値がそのまま残っています。
extern	他のファイルで定義されているグローバル変数を使う際に使用します。
register	register は変数の記憶領域をCPUのレジスタに可能な限り割り当てようとする変数です。
typedef	C言語の構文としては記憶クラス指定子に分類されますが、型に別名を付けるため指定子です。

変数

型修飾子

型修飾子は、その変数の性質を指定するキーワードです。

const	その変数が書き換えできないことを示します。 宣言を行う場合には、初期値が必要です。組み込み用途のコンパイラの場合、ROMに置く変数の宣言に使用する場合もあります。
volatile	その変数へのアクセスについて、コンパイラによる最適化を抑止することを示します。組み込み用途の場合、アクセスの順序をコンパイラの最適化で変更されては困る場合などに使用します。

変数

型指定子(1)

型指定子は、その変数のデータの大きさや構造を指定するキーワードです。

整数型

整数を扱うための型です。C言語の規格では、以下のことのみ決まっています。対象となるCPUのアーキテクチャとC言語の仕様により、具体的なデータ長が決まります。

int	対象となるCPUが自然に扱える大きさのデータです。
short int	int型と同じか、それより小さいデータです。
long int	int型と同じか、それより大きいデータです。
char	文字を格納できる大きさのデータです。ASCIIコードであれば8ビットになります。

変数宣言

型指定子(2)

signed / unsigned

整数型の前後に付加することによって符号の有無を指定します。

省略時には、signed と解釈されます。ただし、一部のコンパイラでは、char型のみ unsigned と解釈されるものもあります。

弊社CISCマイコンの場合

型	表現範囲	データ長
signed char	-128 ~ 127	1バイト
unsigned char	0 ~ 255	1バイト
signed int	-32768 ~ 32767	2バイト
unsigned int	0 ~ 65535	2バイト
signed long	-2147483648 ~ 2147483647	4バイト
unsigned long	0 ~ 4294967295	4バイト

※これらの値は、それぞれのコンパイラのlimit.hで定義されています。

変数

型指定子(3)

浮動小数点型 float / double / long double

小数点を表現するための型です。32ビット単精度浮動小数点数(float)と64ビット倍精度浮動小数点数(double)が用意されています。

弊社CISCマイコンの場合

型	表現範囲	データ長
float	3.4e-38 ~ 3.4e+38 有効桁 7 桁	4バイト
double	1.7e-308 ~ 1.7e+308 有効桁15桁	8バイト

構造体・共用体・列挙体 struct / union / enum

これらは、タグ名を伴ってユーザ定義の型を定義するのに使用します。

配列・ポインタ変数

配列は、同じ型の連続したデータを扱うための型です。ポインタはアドレスを扱うための型です。

変数

識別子(1)

データを操作する際に用いる領域の名前で、変数名ともいいます。

変数名には自由な名前を付けることができますが、関連性のある名前を付けるとプログラム上見やすくなります。また、次のいくつかの点について注意が必要です。

- ①英字、数字、アンダースコアが使用可能
- ②先頭の文字には数字は使用できない
- ③変数名に英字を使用した場合、大文字と小文字は区別される
- ④予約語は使用不可
- ⑤同じスコープ内では、同じ識別子は使用できない

変数

変数の種類

C言語の変数には、通常の変数とは別に、以下のようなものがあります。

これらは、複数の変数をひとまとめにしたり、アドレスを操作するために使用します。

- 1) 「配列」
- 2) 「ポインタ」
- 3) 「構造体」
- 4) 「共用体」

変数

配列(1)

通常の変数には1つのデータしか格納できません。同じ型の連続したデータを取り扱う場合に使用する変数が配列です。

配列宣言

```
型 配列名 [データ数] ;
```

型や配列名に関しては通常の変数の宣言と同じですが、まとめて扱うデータを[]内に宣言する点が異なります。

なお、配列の各要素の各要素を操作する場合には、配列名[要素番号]で、何番目のデータを操作するかを指定します。

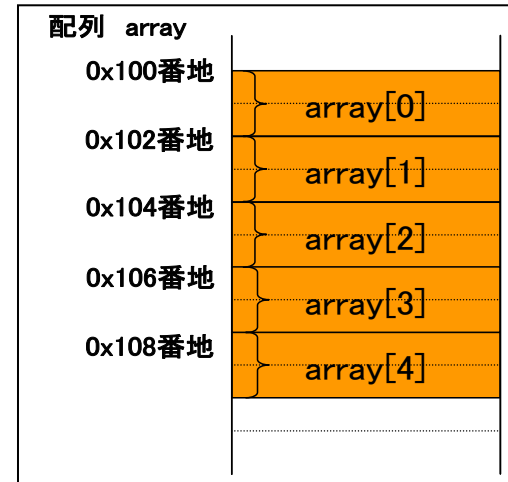
変数

配列(2)

配列宣言と配列の操作の記述例

```
void main(void)
{
    int  array[5];
        . . .
    array[0] = 20;
    array[1] = 30;
        . . .
}
```

配列がメモリに配置されるイメージ図



配列の宣言と同時に初期値を指定することも出来ます。

```
型 配列名 [データ数] = { データ1 , データ2 , . . . } ;
```

変数

配列(3)

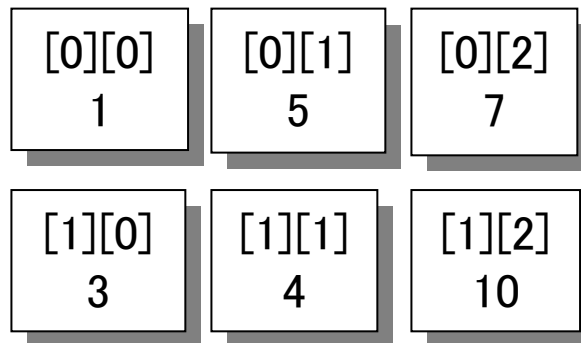
データによっては縦、横、といったように2次元にデータが並んでいる場合もあります。
2次元の配列の宣言は[データ数]を2つ記述します。

型 配列名 [データ数(行)] [データ数(列)] ;

2次元の配列のデータ数は「データ数(行)×データ数(列)」個になります。

2次元の配列のイメージ

```
int array[2][3] = {  
    { 1, 5, 7 },  
    { 3, 4, 10 }  
};
```



変数

(2) ポインタ

メモリ上の変数が、大きい場合(配列や構造体など)は、変数全体を扱うより、変数の先頭アドレスだけ扱う方が効率がよい場合があります。アドレスを取り扱う場合に使用する変数がポインタです。

ポインタ宣言

型 *ポインタ(変数)名 ;

ポインタ変数の型には間接的に扱う(ポインタが指す先の)データの型を指定します。

これは、ポインタが指す先の変数を演算に使用する場合、その変数の型の情報が無いと、正しい結果が得られなくなるためです。

変数

ポインタ(1)

ポインタを使って演算を行うための演算子が2種類あります。

演算子名	演算子	記述例	機能
アドレス演算子	&	&変数名	変数のアドレスを取得する
ポインタ演算子	*	*ポインタ変数名	アクセス先のデータを操作する

```
int  val;    /* int型の変数 */
int  *ptr;   /* int型のポインタ変数 */

ptr = &val;  /* &演算子で変数valのアドレス取得し、ptrに格納 */
*ptr = 100;  /* ptrに格納されたアドレス(&val)の指す変数(val)を100にする */
```

上の例では、int型の変数valにポインタを使って100を設定しています。

変数

ポインタ(2)

配列は、ポインタでも操作することが出来ます。
以下はデータ書き換えを、ポインタで行う例です。
array[5]のデータを、{8,9,10,11,12}に書き換えています。

```
int    array[5] = {1, 2, 3, 4, 5};  
  
void   sub(void)  
{  
    int *ptr = array;  
    *ptr = 8;  
    *(ptr+1) = 9;  
    *(ptr+2) = 10;  
    *(ptr+3) = 11;  
    *(ptr+4) = 12;  
}
```

int型が2バイトである場合、
ポインタ変数 ptr を1進めると、
番地は2進みます

ポインタ変数 ptr を1つつ進めながら、
配列を書き換えていきます。

変数

構造体(1)

異なるデータ型の変数をまとめて扱うものを構造体といいます。
構造体は、既存のデータ型を組み合わせることによって新しいデータ型を定義します。構造体でまとめられた個々のデータはメンバと呼ばれます。

構造体の宣言

```
struct    タグ名 {  
          メンバ1のデータ型 メンバ1の名前;  
          メンバ2のデータ型 メンバ2の名前;  
        } ;
```

構造体の宣言では、
構造体の形だけ決めます。
変数の宣言は別に必要です。

ここで宣言されたタグ名は構造体名ともよばれ、新しいデータ型になります。

構造体の変数宣言

```
struct    構造体名    構造体の変数名;
```

変数の種類

構造体(2)

構造体を操作するための演算子が2種類あります。

演算子名	演算子	機能
ドット演算子	.	構造体のメンバを変数として操作する
アロー演算子	->	構造体変数をポインタ操作する

ドット演算子

変数名で示す構造体のメンバに値を代入します。

構造体の変数名.メンバ名 = 代入する値 ;

アロー演算子

ポインタが指し示すメンバの先頭アドレスに値を代入します。

ポインタ名 -> メンバ名 = 代入する値 ;

変数の種類

構造体(3)

構造体の宣言及び操作例を示します。

```
struct s_data{  
    int s_mem1;  
    unsigned char s_mem2;  
};  
void sub(void)  
{  
    struct s_data sval;  
  
    struct s_data *sptr = &sval;  
  
    sval.s_mem1 = 100;  
  
    sptr->s_mem2 = 200;  
}
```

構造体の宣言

構造体変数の宣言

構造体s_dataを指すポインタ変数の宣言
初期値は svalのアドレス

構造体変数を使ったメンバの操作

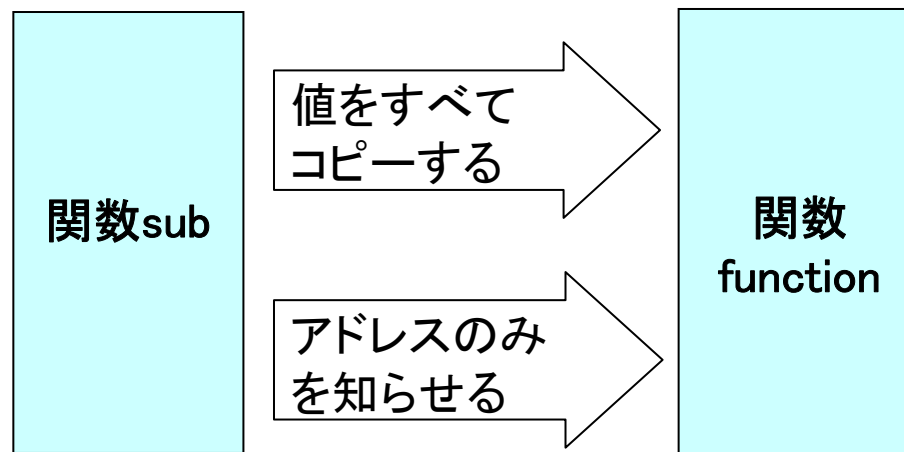
構造体を指すポインタ変数を使った
メンバの操作

変数の種類

構造体(4)

2つ以上の関数で、同じ構造体変数进行操作する方法には、以下のようなものがあります。場合に応じて使い分けてください。

- ・グローバルで構造体変数を宣言しておき、すべての関数から直接操作する方法
- ・ローカルで構造体変数を宣言しておき、関数間で構造体変数自体を受け渡す方法（構造体変数の値渡し）
- ・ローカルで構造体変数を宣言しておき、関数間で構造体変数のアドレスを受け渡す方法（構造体変数のポインタ渡し）



変数の種類

構造体(5)

構造体変数の値渡し

```
struct s_data{
    unsigned char array[3];
    unsigned int data;
};
struct s_data function(struct s_data f1_val);
void sub( void )
{
    static struct s_data    val;
    val.array[0] = 3;
    val.array[1] = 2;
    val.array[2] = 1;
    val.data = 0;
    val = function(val);
    val.array[0]++;
    val.array[1]++;
    val.array[2]++;
    val.data++;
    while(1);
}
struct s_data function( struct s_data f1_val)
{
    f1_val.array[0] = 4;
    f1_val.array[1] = 5;
    f1_val.array[2] = 6;
    f1_val.data = 7;
    return f1_val;
}
```

構造体変数で
メンバへアクセス。
戻り値は
構造体変数。

構造体変数のポインタ渡し

```
struct s_data{
    unsigned char array[3];
    unsigned int data;
};
void function(struct s_data *pf1_val);
void sub( void )
{
    static struct s_data    val;
    val.array[0] = 3;
    val.array[1] = 2;
    val.array[2] = 1;
    val.data = 0;
    function(&val);
    val.array[0]++;
    val.array[1]++;
    val.array[2]++;
    val.data++;
    while(1);
}
void function( struct s_data *pf1_val)
{
    pf1_val->array[0] = 4;
    pf1_val->array[1] = 5;
    pf1_val->array[2] = 6;
    pf1_val->data = 7;
}
```

構造体を指すポイン
タ変数を使用し、
メンバへアクセス。
戻り値はなし。

変数の種類

共用体(1)

異なるデータ型の変数をまとめて扱うものを共用体といいます。
構造体との違いは、「別のメンバが同じメモリ領域を共有する」点にあります。

共用体の宣言

```
union タグ名 {  
    メンバ1のデータ型 メンバ1の名前;  
    メンバ2のデータ型 メンバ2の名前;  
};
```

ここで宣言されたタグ名は共用体名ともよばれ、新しいデータ型になります。

共用体の変数宣言

```
union 共用体名 共用体の変数名;
```

変数の種類

共用体(2)

共用体进行操作するための演算子が2種類あります。

演算子名	演算子	機能
ドット演算子	.	共用体のメンバを変数として操作する
アロー演算子	->	共用体変数をポインタ操作する

ドット演算子

変数名で示す共用体のメンバに値を代入します。

共用体の変数名. メンバ名 = 代入する値 ;

アロー演算子

ポインタが指し示すメンバの先頭アドレスに値を代入します。

ポインタ名 -> メンバ名 = 代入する値 ;

変数の種類

共用体(3)

共用体の宣言及び操作例を示します。

```
union  u_data{  
    unsigned int    u_mem1;  
    unsigned char   u_mem2[2];  
};  
  
void  sub(void)  
{  
    union u_data  uval;  
    union u_data  *uptr = &uval;  
    uval.u_mem1 = 0x0009;  
    uptr->u_mem2[0] = 0x07;  
}
```

共用体の宣言

共用体変数の宣言

共用体u_dataを指すポインタ
変数の宣言
初期値は uvalのアドレス

共用体変数を使ったメンバ
の操作

共用体を指すポインタ変数
を使ったメンバの操作

変数の種類

さまざまな変数の例

```
signed int    i, j;
```

符号付int型変数
で区切るにより、同じ型の変数を
宣言できる

```
unsigned long int ul;
```

符号無しlong型変数

```
char    c1[5] = "abcde";
```

文字型配列
初期値として文字列を指定
文字列は "で括る

```
const double    d = 12.34;
```

書き換え不能の初期値付き倍精度浮動
小数点数
初期値は数値

```
struct one_two_type{  
    unsigned char uc1;  
    unsigned int  ui1;  
} o_t_1;
```

構造体の宣言
1バイトの符号無し と 2バイトの符号無し
でセット

```
union two_one_type{  
    unsigned char uca2[2];  
    unsigned int  ui2;  
} t_o_1;
```

配列の宣言
1バイトの符号無しを2つセットで扱う

共用体の宣言
同じメモリを配列と通常の変数で使い
分ける

関数

関数

関数は、一連の処理を行うために必要な命令をまとめたものです。

関数を使うメリットは 処理をまとめる事で、同じ処理を何度も記述しなくて済むようになります。また、機能ごとにまとめる事で、プログラムを読みやすくしたり、保守をしやすくする事もできます。

一旦まとめた関数は、「呼び出す」ことにより実行します。

関数定義

```
記憶クラス 関数の型 関数名(引数の型1 引数名1, 引数の型2 引数名2, . . .)
{
    .
    .
    .
}
```

関数の範囲は{ から }までになります。{ }の中には実行する処理を記述し、返却値がある場合はreturn文を記述します。

関数

関数の型と記憶クラス

記憶クラス

記憶クラスにはstaticとexternがあり、記憶クラスを省略した場合、自動的にextern関数として認識されます。また、staticを使用すると、他のファイルからは使用することができなくなります。

関数の型

関数の型はreturn文で返すデータ(返り値)の型を記述します。返り値がない場合、型にはvoidを記述します。

```
static void func1(void)
{
    /* void は戻り値がないという意味 */
}

int func1(void)
{
    /* 記憶クラスが省略されているのでextern */
}
```

記憶クラス static の
戻り値のない関数

記憶クラス extern の
int 型を返す関数

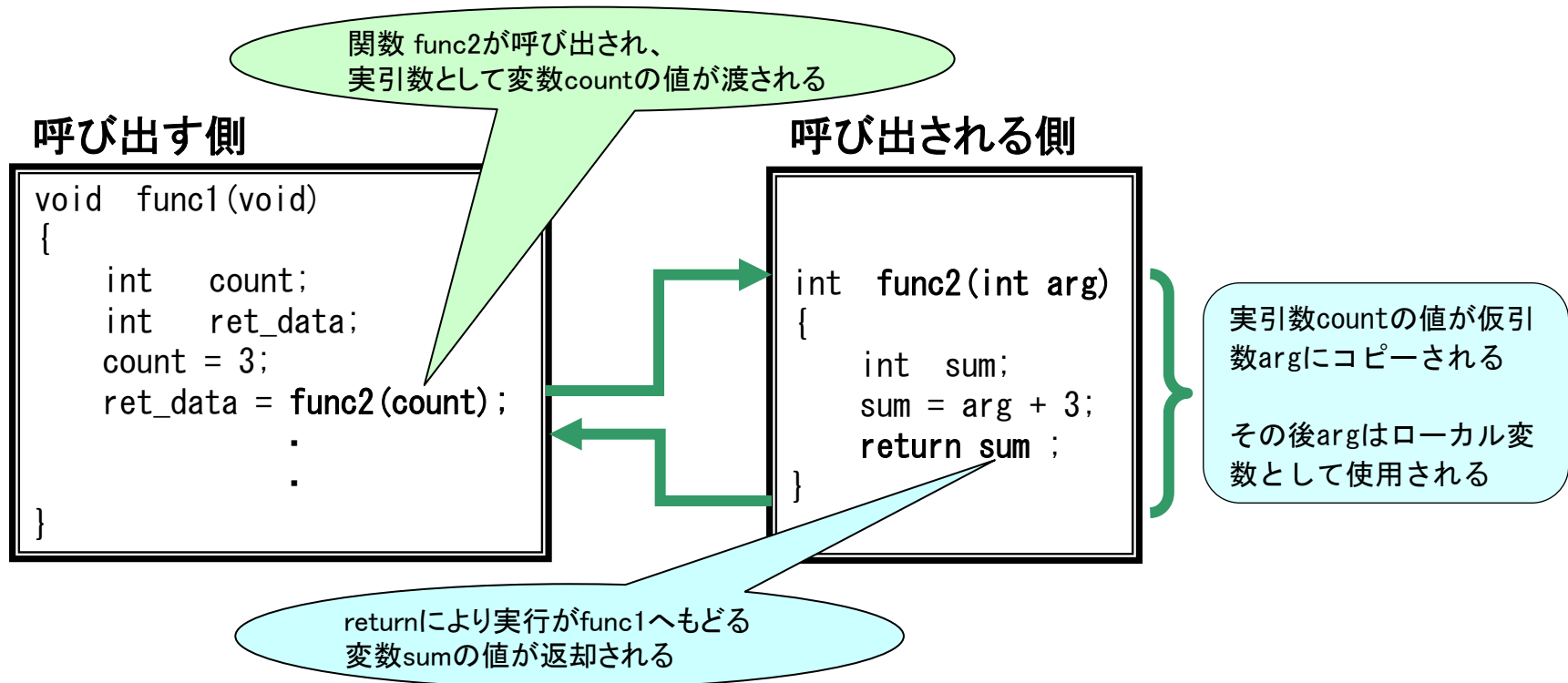
関数

引数(argument)

引数は、呼び出す関数に処理をさせるために渡すデータです。引数には以下のものがあります。

実引数: 関数を実際に使用するときに取り渡された引数

仮引数: 関数定義時に使用される引数



関数

プロトタイプ宣言

関数呼び出しの前に、あらかじめ関数の型や引数についての情報を宣言しておく必要があります。これを「プロトタイプ宣言」といいます。

プロトタイプ宣言には関数の型、関数名、引数の数、引数の型が必要です。

<main.c>

```
//プロトタイプ無し  
main()  
{  
    char c1;  
    c1=func(100, 1);  
}
```

プロトタイプ宣言無しでは、
どのような関数かわからずに
呼び出してしまうので、
正しく動作しないことがある

<sub.c>

```
char func(int a, char b)  
{  
    //  
    return 1;  
}
```

```
char func(int, char);  
main()  
{  
    char c1;  
    c1=func(100, 1);  
}
```

プロトタイプ宣言があれば、
正しく関数を呼び出す
ことができる

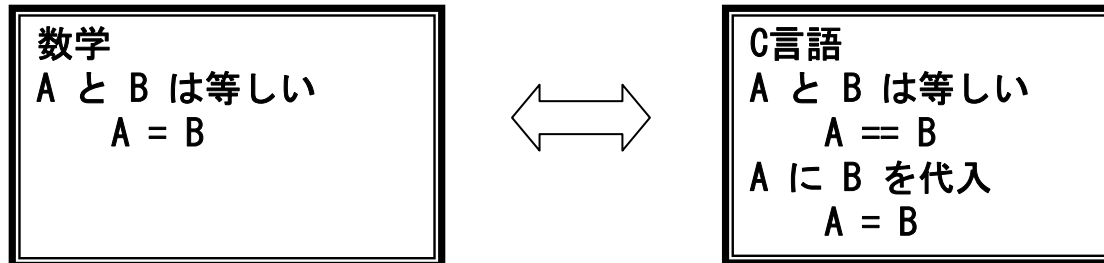
```
char func(int a, char b)  
{  
    //  
    return 1;  
}
```

演算子

演算子

C言語と数学の演算子の違い(1)

C言語では、記号で演算を指示するものを演算子と呼び、数学の数式に近い意味を持っています。ただし、一部の演算子には、数式と異なる記号が用いられたり、数学の演算子とは異なる動作をするものもあります。



数学の演算子と異なる例を以下に示します。

```
/* 数学とは異なる記号を使用する演算子 */
a = b * 3;          /* b掛ける3の値をaへ代入 */
a = b / 3;          /* b割る3の商をaへ代入 */
a = b % 3;          /* b割る3の余りをaへ代入 */

/* 前置増分演算子 */
a = 1;
b = ++a;            /* aを先に1増やしてから、bへ代入 */

/* 後置増分演算子 */
a = 1;
b = a++;            /* aをbへ代入してから、1増やす */
```

演算子

C言語と数学の演算子の違い(2)

ある変数bの値が、aより大きくcより小さい場合、数学では以下のように表現します。

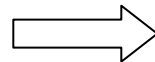
$$a < b < c$$

C言語では、以下のように記述する必要があります。

$$a < b \ \&\& \ b < c$$

C言語の関係演算子 ' $<$ ' は、左オペランドを評価した値が、右オペランドを評価した値より小さければint型の1を、そうでなければint型の0を生成します。

$$\begin{array}{c} a < b < c \\ \downarrow \\ (a < b) < c \text{ と認識} \end{array}$$



aとbの 関係	(a<b)の 評価	(a<b)<cの 展開	cの値	a<b<c の評価
a<b	1	1 < c	c > 1	1
			c ≤ 1	0
a ≥ b	0	0 < c	c > 0	1
			c ≤ 0	0

制御文

制御文

制御文には以下のような種類があり、大きく分けると、条件判定によって実行する処理を選択する「選択文」と、条件が成立している間、同じ処理を行う「繰り返し文」があります。

選択文

1)「if 文」

2)「switch 文」

繰り返し文

3)「for 文」

4)「while 文」

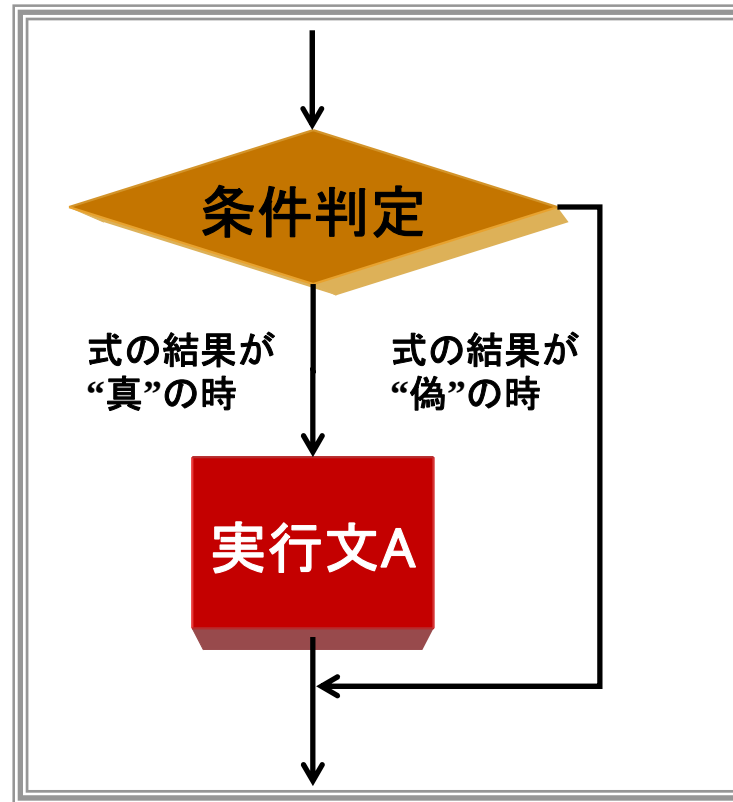
5)「do-while 文」

制御文

if文(1)

if文は条件判定によって2方向(真or偽)に分岐され、それぞれに続く処理が実行されます。条件文には関係演算子や論理演算子を使用し、真か偽かを判定します。

```
if(条件) {  
    実行文A;  
    .  
    .  
    .  
}
```

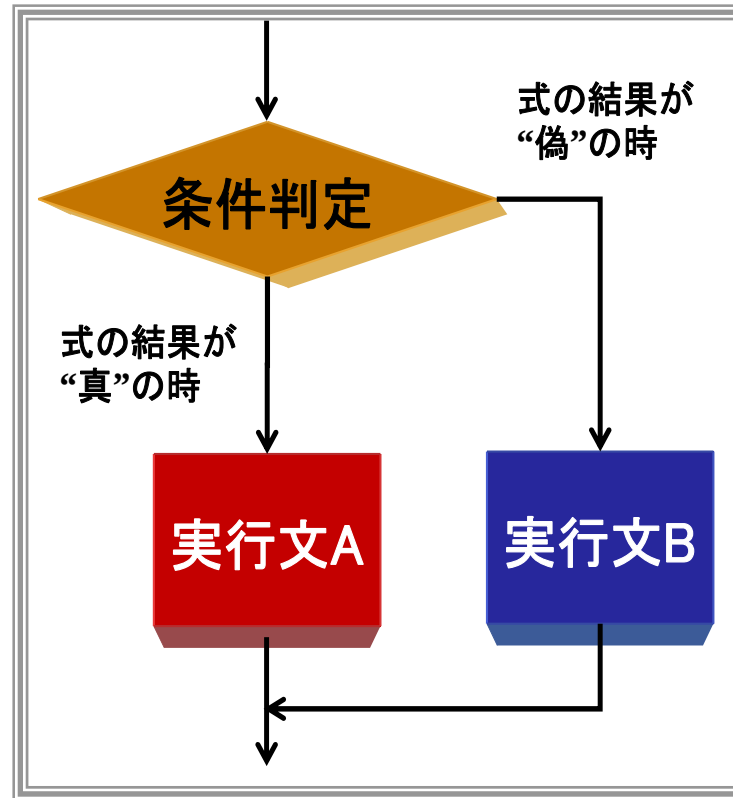


制御文

if文(2)

条件判定によって2方向(真or偽)に分岐され、それぞれ別の処理を実行する際には、次のように記述します。

```
if(条件) {  
    実行文A;  
    .  
}  
else {  
    実行文B;  
    .  
}
```



制御文

switch文(1)

switchの後の()の中に記述される1つの式によって多方向分岐を行う制御文です。
式に対応するcaseが実行されます。

switch文を記述するときには以下のいくつかの点について注意が必要です。

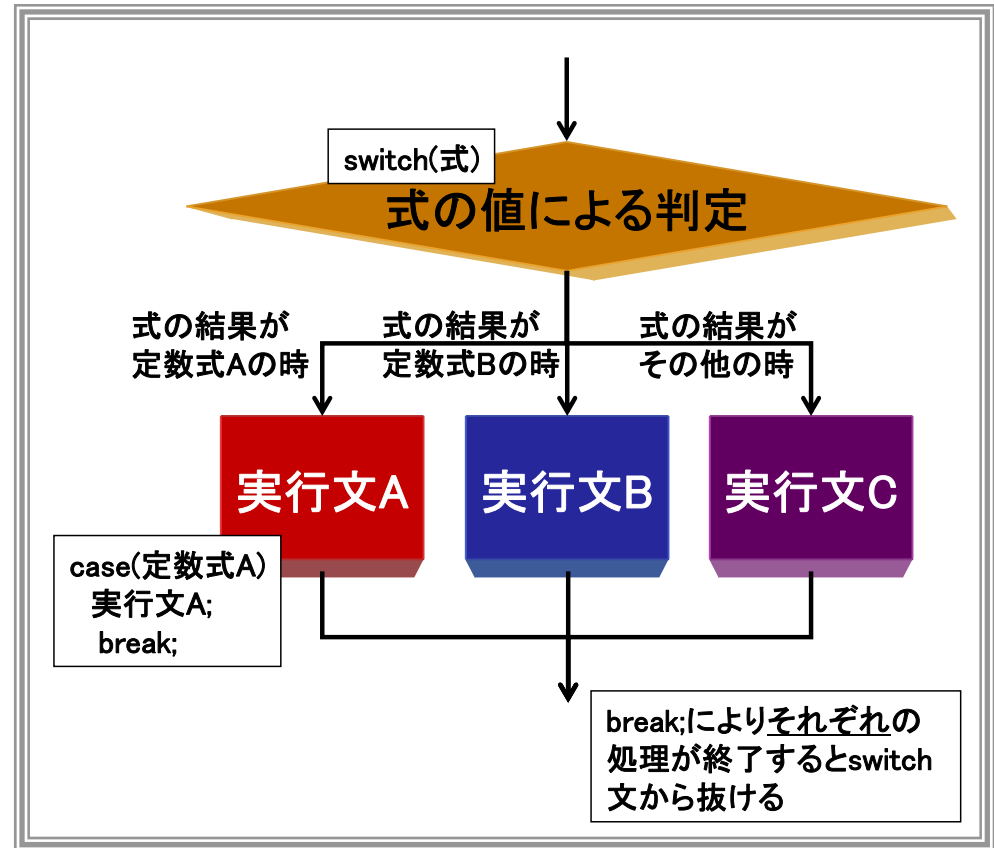
- ① case()のあとにはコロンをつけます。
- ② default文はどのcase文にも該当しない場合に実行するブロックです。
- ③ case文のあとには複数の文を記述できます。
- ④ break文を記述しない場合、実行している処理より以下にかかっているプログラムを実行します。

制御文

switch文(2)

条件判定によって3方向(定数式A,定数式B,それ以外)に分岐され、それぞれ別の処理を実行する際には、次のように記述します。

```
switch(式) {  
    case(定数式A):  
        処理A;  
        break;  
    case(定数式B):  
        処理B;  
        break;  
    default:  
        処理C;  
}
```



制御文

for文(1)

繰り返す回数が決まっている場合にfor文を使用します。for文ではカウンタ変数を使用し、繰り返し回数の制御を行います。

```
for( 代入式 ; 条件式 ; 増減式 ) {  
    処理  
    .  
}
```

代入式: カウンタ変数の初期値を設定
条件式: カウンタ変数の終了条件を設定
増減式: カウンタ変数の増減値を設定

初期値、条件式、増減式を省略した場合、無限ループが生成されます。

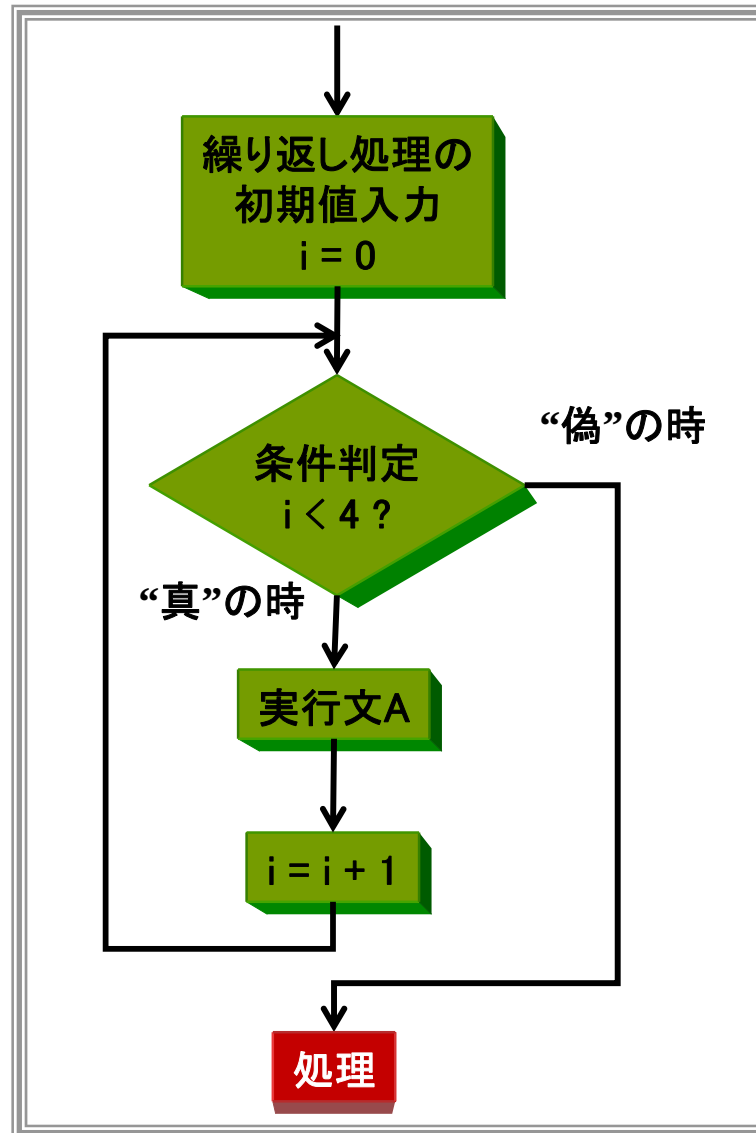
```
for( ; ; ) {  
    処理  
    .  
}
```

制御文

for文(2)

4回実行文Aの内容を実行し、5回目には実行文Aを実行せず次の処理に移動する例を以下に示す。

```
for ( i = 0 ; i < 4 ; i++ ) {  
    data = data+3 ;  
    data2 = data ;  
}
```

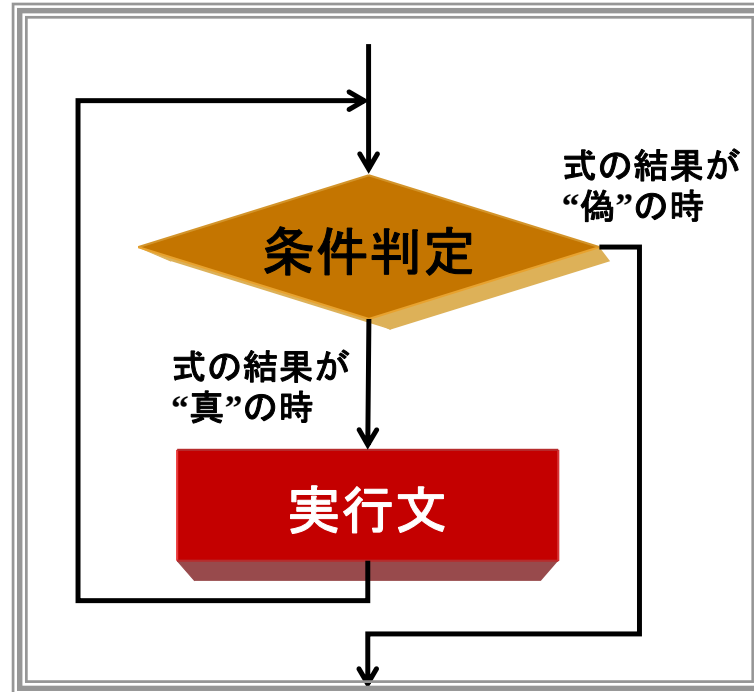


制御文

while文

繰り返す回数が不定である場合に使用します。

```
while(条件式) {  
    処理  
    .  
}
```



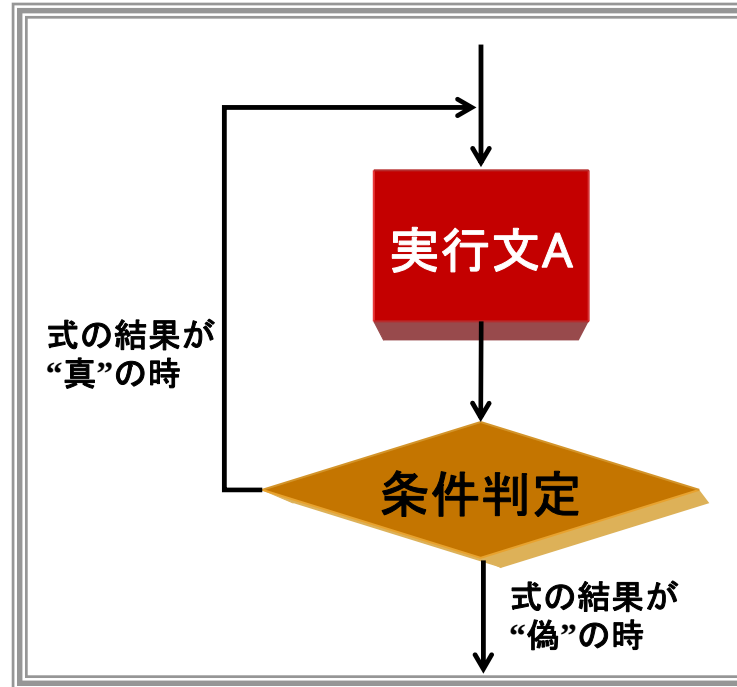
処理を行う前に条件判定を行います。
条件式に1を記述した場合、無限ループになります。

制御文

do-while文

繰り返す回数が不定である場合に使用します。

```
do {  
    処理  
    .  
} while(条件式) ;
```



処理を行ってから条件判定を行います。
つまり、条件の真/偽に関わらず、
最低一度は処理を実行したい場合に用います。

制御文

繰り返し文のポイント

繰り返し文には、for, while, do-whileの3種類があり、いずれの方法でも記述できる場合がありますが、その使い分けにプログラムで一環としたルールを設けると、読みやすくなります。

たとえば、以下のようなルールがあげられます。

ループ回数をカウンタの変数で表現できる	→ for文
ループ回数をカウンタの変数で表現できない、または複雑	→ while文

C言語入門のまとめ

各項目のポイント

－C言語とは

- C言語は移植性が高く、保守しやすいプログラミング言語！

－プリプロセス

- プリプロセスを活用すれば、修正しやすいソースファイルに！

－変数

- スコープと記憶域期間で、変数の性質が決まる！

－関数

- プロトタイプ宣言は、正しい関数呼び出しのために必要！

－演算子

- 数学と似て非なる記号の演算子に注意！

－制御文

- 繰り返し文は、ループカウンタの種類で選ぶ！

C言語

- 本資料に掲載されているハードウェア、ソフトウェアおよびシステム（以下、本製品という）に関する情報等、本資料の掲載内容は、技術の進歩などにより予告なしに変更されることがあります。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。また、文書による当社の事前の承諾を得て本資料を転載複製する場合でも、記載内容に一切変更を加えたり、削除したりしないでください。
- 当社は品質、信頼性の向上に努めていますが、半導体製品は一般に誤作動または故障する場合があります。本製品をご使用頂く場合は、本製品の誤作動や故障により生命・身体・財産が侵害されることのないように、お客様の責任において、お客様のハードウェア・ソフトウェア・システムに必要な安全設計を行うことをお願いします。なお、設計および使用に際しては、本製品に関する最新の情報（本資料、仕様書、データシート、アプリケーションノート、半導体信頼性ハンドブックなど）および本製品が使用される機器の取扱説明書、操作説明書などをご確認の上、これに従ってください。また、上記資料などに記載の製品データ、図、表などに示す技術的な内容、プログラム、アルゴリズムその他応用回路例などの情報を使用する場合は、お客様の製品単独およびシステム全体で十分に評価し、お客様の責任において適用可否を判断してください。
- 本製品は、一般的電子機器（コンピュータ、パーソナル機器、事務機器、計測機器、産業用ロボット、家電機器など）または本資料に個別に記載されている用途に使用されることが意図されています。本製品は、特別に高い品質・信頼性が要求され、またはその故障や誤作動が生命・身体に危害を及ぼす恐れ、膨大な財産損害を引き起こす恐れ、もしくは社会に深刻な影響を及ぼす恐れのある機器（以下“特定用途”という）に使用されることは意図されていませんし、保証もされていません。特定用途には原子力関連機器、航空・宇宙機器、医療機器、車載・輸送機器、列車・船舶機器、交通信号機器、燃焼・爆発制御機器、各種安全関連機器、昇降機器、電力機器、金融関連機器などが含まれます。本資料に個別に記載されている場合を除き、本製品を特定用途に使用しないでください。
- 本製品を分解、解析、リバースエンジニアリング、改造、改変、翻案、複製等しないでください。
- 本製品を、国内外の法令、規則及び命令により、製造、使用、販売を禁止されている製品に使用することはできません。
- 本資料に掲載してある技術情報は、製品の代表的動作・応用を説明するためのもので、その使用に際して当社及び第三者の知的財産権その他の権利に対する保証または実施権の許諾を行うものではありません。
- 別途、書面による契約またはお客様と当社が合意した仕様書がない限り、当社は、本製品および技術情報に関して、明示的にも黙示的にも一切の保証（機能動作の保証、商品性の保証、特定目的への合致の保証、情報の正確性の保証、第三者の権利の非侵害保証を含むがこれに限らない。）をしておりません。
- 本製品にはGaAs（ガリウム砒素）が使われているものがあります。その粉末や蒸気等は人体に対し有害ですので、破壊、切断、粉碎や化学的な分解はしないでください。
- 本製品、または本資料に掲載されている技術情報を、大量破壊兵器の開発等の目的、軍事利用の目的、あるいはその他軍事用途の目的で使用しないでください。また、輸出に際しては、「外国為替及び外国貿易法」、「米国輸出管理規則」等、適用ある輸出関連法令を遵守し、それらの定めるところにより必要な手続を行ってください。
- 本製品には、外国為替及び外国貿易法により、輸出または海外への提供が規制されているものがあります。
- 本製品のRoHS適合性など、詳細につきましては製品個別に必ず弊社営業窓口までお問合せください。本製品のご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用ある環境関連法令を十分調査の上、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は一切の責任を負いかねます。
- 上記に加えて、以下は開発ツールのみに適用されます。
- 当社は品質、信頼性の向上に努めていますが、本製品は誤作動または故障する場合があります。本製品をご使用頂く場合は、本製品の誤作動や故障により生命・身体・財産が侵害されることのないようにご使用ください。本製品をご使用頂く場合は、本製品に関する最新の情報（本資料、取扱説明書、仕様書、データシートなど）をご確認の上、これに従ってください。
- 本製品は、半導体製品の機能評価に使用されることを意図しています。機能評価以外の目的（温度・湿度特性評価、信頼性評価など）には使用しないでください。
- 本製品をお客様の製品に組み込まないでください。また、本製品を販売、譲渡、貸与等しないでください。